Slide 1

Administrivia

- Next homework coming soon . . .

Slide 2

Another Way to Get Input — Command-Line Arguments

- Now that we know about arrays, pointers, and text strings, we can talk about command-line arguments. What are they? text that comes after the name of the program on the command line (e.g., when you write `gcc -Wall myprogram.c`, there are are two command-line arguments), possibly modified by the shell (e.g., for filename wildcards).
- Most programming languages provide a way to access this text, often via some sort of argument to the main function/method.

Slide 3

Command-Line Arguments in C

- In C, command-line arguments are passed to `main` as an array of text strings. So if you define `main` as

  ```
  int main(int argc, char * argv[]) { .... }
  ```

  `argc` is the number of arguments, plus one, and `argv` is an array of strings containing the arguments.

  ("Plus one"? yes, `argv[0]` is something system-dependent, often the path for the program's executable.)

- What if you want to get numeric input? you must convert string pointed to by `argv[i]` to the type you want, e.g., with `atoi` or `strtol`.

Slide 4

Command-Line Arguments and UNIX Shells

- Be aware that most UNIX shells do some preliminary parsing and conversion of what you type — e.g., splitting it up into "words", expanding wildcards, etc., etc.
- If you don't want that — enclose in quotation marks or use escape character (backslash).

Slide 5

Character-Oriented I/O in C

- Two useful functions to know about: `getchar` and `putchar`.
- Both treat characters as integers (which is allowed). `getchar` returns a special value, `EOF`, at "end of file". How to signal this when standard input is from keyboard is system-dependent — often(?) control-D on UNIX-like systems.

Slide 6

I/O in C — Review

- `getchar` and `putchar` provide simple character-at-a-time I/O to standard input/output.
- `printf` and `scanf` provide more sophisticated functionality, but again for standard input/output.
- I/O redirection provides one way to work with files. Is there something more general? Yes, but first review redirection . . .

Slide 7

Sidebar — Input/Output Redirection in UNIX/Linux

- In programming classes I talk about “reading from standard input” rather than “reading from the keyboard”, and “writing to standard output” (or “writing to standard error”) rather than “writing to the screen”.
- What’s the difference?

Slide 8

I/O Redirection, Continued

- `stdin` (standard input) can come from keyboard, file, or from another program or shell script.
- `stdout` and `stderr` (standard output, error) can go to terminal or file (overwrite or append), separately or together.

Slide 9

I/O Redirection, Continued

- For example — to redirect output of `ls` to `ls.out`, type

  `ls >ls.out`

  (Overwrites `ls.out`. To append, replace > with >>.)

  To also redirect any error messages, append `2>&1`.
- To redirect input, use `<infile`.

Slide 10

File I/O — Streams

- C's notion of file I/O is based on the notion of a *stream* — a sequence of characters/bytes. Streams can be *text* (characters arranged into lines separated by something platform-dependent) or *binary* (any kind of bytes). UNIX/Linux doesn't make a distinction, but some other operating systems do.
- An input stream is a sequence of characters/bytes coming into your program (think of characters being typed at the console).
- An output stream is a sequence of characters/bytes produced by your program (think of characters being printed to the screen, including special characters such as the one for going to the next line).

Slide 11

Streams in C

- In C, streams are represented by the type `FILE *` — i.e., a pointer to a `FILE`, which is something defined in `stdio.h`.
- A few streams are predefined — `stdin` for standard input, `stdout` for standard output, `stderr`) for standard error (also output, but distinct from `stdout` so you can separate normal output from error messages if you want to).
- To create other streams . . .

Slide 12

Creating Streams in C

- To create a stream connected with a file — `fopen`.
- Parameters, from its `man` page:
  - First parameter is the name of the file, as a C string.
  - Second parameter is how we want to access the file – read or write, overwrite or append — plus a `b` for binary files, also a string.
  - Return value is a `FILE *` — a somewhat mysterious thing, but one we can pass to other functions. If `NULL`, the open did not succeed. (Can you think of reasons this might happen?)

Slide 13

Working With Streams in C

- To read from an input stream — `fscanf`, almost identical to `scanf`. To write to an output stream — `fprintf`, almost identical to `printf`. `fgetc` and `fputc` provide single-character input and output.
- When done with a stream, `fclose` to tidy up. (Particularly important for output files, which otherwise may not be completely written out.)

Slide 14

Reading Text Strings

- Getting text-string input is surprisingly tricky. `scanf` (or `fscanf`) seems like an obvious choice, but:
  - it can't read a string that includes blanks, and
  - it has no nice way to limit the number of characters read to the size of the array being read into.

  .
- Getting a whole line is probably better. `gets()` is an obvious/simple choice for reading from standard input, but it also has no way to limit how much is read. `fgets()` is better. (Look at its `man` page.)

  (Also notice `puts()` — simple way to write out a text string.)
- (Why do you care about limiting how much is read? not doing so can crash your program or even represent a security risk . . .)

Slide 15

Simple Examples

- (Examples as time permits.)

Slide 16

Minute Essay

- None — sign in.